1984

Towards a Distributed File System

Walter F. Tichy

Zuwang Ruan

Report Number:
84-480

TOWARDS A DISTRIBUTED FILE SYSTEM

Walter F. Tichy
Zuwang Ruan

CSD-TR-480
May 1984

Towards a Distributed File System

Tilde Report CSD-TR-480

May 14, 1984

Walter F. Tichy
Zuwang Ruan

ABSTRACT

This paper describes IBIS, a distributed file system for a network of UNIX machines. IBIS provides two levels of abstraction: file access transparency and file location transparency. File access transparency means that all files are accessed in the same way, regardless of whether they are remote or local. File location transparency hides the location of files in the network. IBIS provides a single, location transparent, hierarchical file system that spans several machines. IBIS exploits location transparency by replicating files and migrating them where they are needed. Replication and migration improve file system efficiency and fault tolerance.

This paper reports on the design, implementation, and performance of the access transparency level, and describes the design of the location transparency level.

This project is supported in part by grants from the National Science Foundation (MCS-8219178) and SUN Microsystems Incorporated.

1. Introduction

IBIS is a distributed file system for a network of UNIX machines. The purpose of IBIS is to provide a uniform, UNIX-compatible file system that spans all nodes in a network. The novel features of IBIS are file migration and replication, which improve the efficiency of file access by placing files near the network nodes where they are used. IBIS is also implemented with a new, decentralized protocol for file access. A forerunner of IBIS is STORK [1], which demonstrates the feasibility of file migration. IBIS is being built as part of the TILDE project [2], whose goal is to integrate a cluster of machines into a single, large computing engine by hiding the network.

IBIS has 2 levels of abstraction. The first level provides *file access transparency* (also called *system call transparency*). File access transparency is achieved if file access primitives like *open*, *close*, *read*, and *write* operate on any file in the network, regardless of the location of the file. Access transparency hides the access method for remote files; it does not hide the location of files.

The second level of abstraction provides *file location transparency*, which frees the user from remembering at which host a file is located. Placement of files becomes the responsibility of the file system, permitting it to exploit placement strategies that improve efficiency. *File migration* is a strategy that improves access time by placing files near the nodes where they are read and written. *File replication* is a strategy that creates copies of files to improve the efficiency of read accesses as well as the fault tolerance of the file system.

The next section describes the access transparency layer of IBIS. Besides presenting design and implementation, we discuss an authentication mechanism that guarantees the security of remote access, and provide performance measurements. Section 3 describes the design of the location transparency layer of IBIS.

2. File Access Transparency

We have extended all UNIX file access primitives to operate on both local and remote files. For example, *open* accepts a filename of the form [<*hostname*>:]<*pathname*>. If <hostname> is missing or the same as the current host, *open* executes a normal system call for opening the file and passes back the returned, local file descriptor. If the file is remote, *open* communicates with a server process on the remote machine to open the desired file remotely. *Open* returns a remote file descriptor in that case. All further operations on the returned file descriptor interact with the local file system or the remote server, as the case may be. Each user process, called a *client*, has its own, dedicated server process at the remote host. The dedicated server and a connection between client and server are created by a special server creation process on the remote host, at the time when the client accesses the first file on that host. If the client accesses several files on the same remote host, the dedicated server is multiplexed among all requests.

IBIS implements all remote file operations with the IPC facilities of UNIX 4.2 BSD, and is based on TCP/IP. The access transparency layer is sandwiched between the UNIX kernel and user programs. For local files, the layer simply invokes the standard UNIX system calls. For remote files, the layer observes a remote procedure call protocol for interacting with the server. Since IBIS uses TCP/IP rather than streamlined protocols, highly reliable operation results. Furthermore, IBIS is not restricted to local area networks. It can provide reliable file access between any pair of Unix systems running TCP, even if they are connected via gateways and lossy subnets. As will be shown below, the performance penalty for this generality is modest.

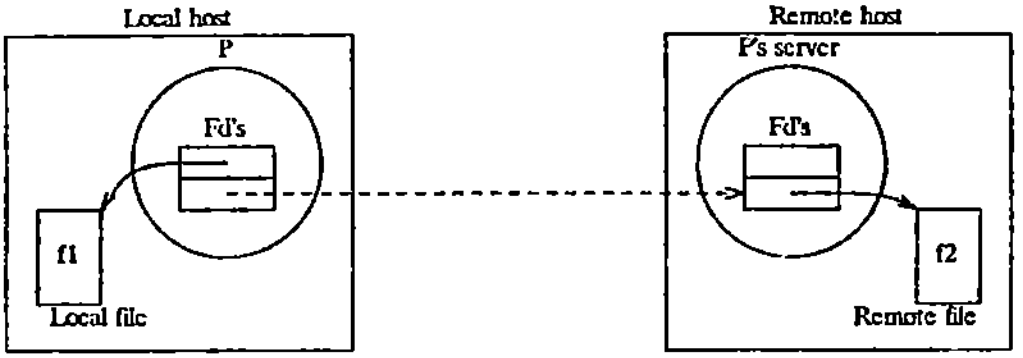
In UNIX, open file descriptors are inherited across the system calls *fork* and *execve*, which create new processes. IBIS guarantees the same behavior for remote file descriptors. For example, *fork* operates as follows (see Fig. 1). If a process with remote file descriptors forks a child process, the server forks a new server for the child on the remote machine. The child server then connects to the child process. The child server automatically shares the open files with the parent server. This protocol results in the same behavior as if the files had been local. Thus, remote file operations simulate the behavior of the UNIX file system exactly.

The complete list of system calls and I/O functions with remote access appears in the appendix. Because of faithful access transparency and a complete set of file operations, almost all existing programs can be upgraded to interact with remote files by simply relinking them with the new I/O functions. As test cases, we have relinked the following UNIX commands: cat, chmod, cp, csh, diff, ed, ln, ls, mkdir, mv, rm, rmdir, and all RCS operations [3]. For example, upgrading the command *ls* for remote access required no special treatment, since *ls* simply opens a directory as a remote file. The command *rm* required a minor modification. The Purdue version of *rm* does not remove a file; instead, it moves the file to a special directory called *tomb*, where it will be deleted after a few days. Simply relinking *rm* with the new library resulted in a program that moved a file to the *tomb* on the host where the command was executed. Thus, *rm* moved all remote files to the local machine, causing large amounts of useless data to be transmitted. A simple modification ensured that *rm* now moves a file to the *tomb* at the file's home machine.

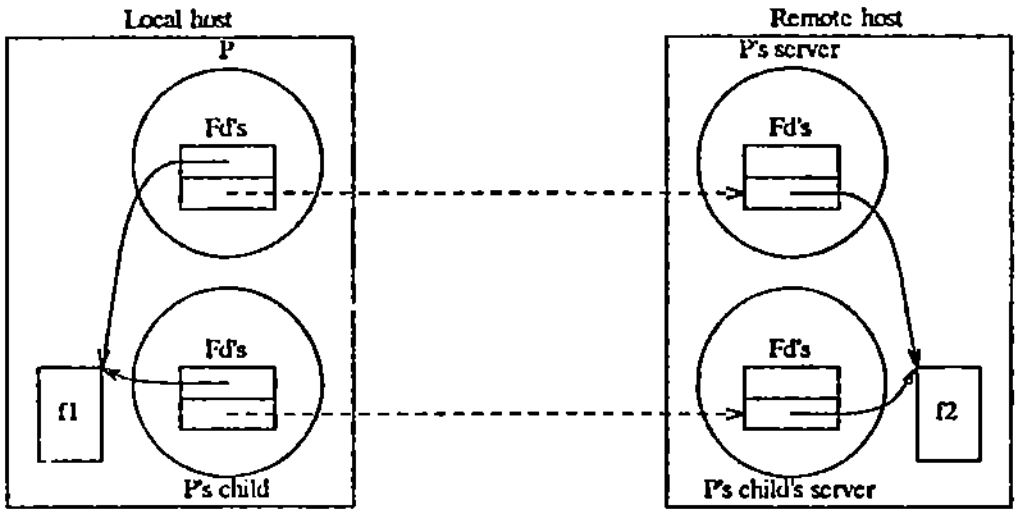
An important application of the remote access primitives is a version of *csh* with remote access, called *rcsh*. Although *rcsh* executes all commands on the local machine, it provides I/O redirection for remote files, and transparent filename substitution on local and remote machines. For example, suppose the command

```
cat  host2 : foo.*  >  host3 : result
```

is executed on *host1*. *Rcsh* first generates a list of filenames by performing filename substitution on *host2*. *Cat* runs on *host1*, but opens the files on *host2* remotely. *Rcsh* then redirects the output of *cat* to the file *result* on *host3*.



a) Before fork



b) After fork

Figure 1: Remote file access before/after fork.

Another convenient shorthand made possible by the remote access primitives are cross-machine symbolic links. For example, the command

```
ln  -s  host1 : path  h1
```

establishes the synonym *h1* for *host1:path*. Whenever a program accesses a file via *h1*, an implicit remote access is performed. With symbolic links crossing machines, a user can construct a directory tree that spans several machines. Remote symbolic links can simulate location transparency (but not replication and migration) to a certain degree.

2.1. Authentication

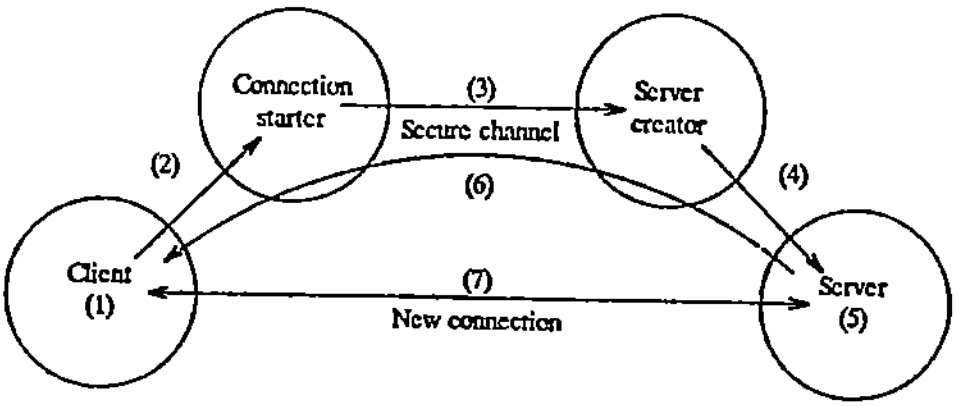
A fundamental problem with remote access is authentication. Remote access should not require additional authentication from the user, yet should be secure enough to prevent impersonation and violation of access rights. For example, while a connection between a client and a server is being established, it is possible that a malicious program impersonates the server and grabs the connection to the client before the real server has a chance to connect. To foil such an attack, the client needs a way of ascertaining the authenticity of the server. Similarly, the server must have a way of determining that it is connected to the right client. Finally, the server must observe the access rights that the client process has. The client process should gain no more nor less access rights through the server than if the client were running directly on the remote machine.

We solved these problems as follows. The dedicated server is established by an authentication mechanism called a *"two-way handshake via secure channel"* (see Fig. 2). When a client process P attempts to establish a connection to a remote host, it reserves a port number A and executes a connection starter program. The connection starter opens a secure channel to the server creator in the remote host and passes the user id of P to it. The server creator then establishes a server Q with the access rights of P, as indicated by the user id. Q reserves a portnumber B. Next, P and Q exchange their portnumbers via the secure channel between the connection starter and server creator. Finally, P and Q establish their own connection, and verify each other's portnumber. This mechanism assures that neither server nor client can be impersonated by another process, and that the server has the correct access permissions. The secure channel is implemented by making both connection starter and server creator privileged processes which communicate via privileged port numbers. Note that the client and its server are not privileged and do not use privileged port numbers. A privileged channel is only used for setting up the dedicated connection. The Unix 4.2 commands *rcp* and *rsh* use the privileged channel continuously, and must therefore be privileged.

2.2. Performance

We compared the execution times of our remote commands with the normal UNIX system calls for local access, and with *rcp* for remote access. Measuring execution times was difficult, since local CPU times do not reflect delays caused by communication with remote servers. We therefore chose to consider only elapsed time. Elapsed time includes delays caused by communication, but also delays caused by timesharing. The measurements were taken with only 1 or 2 users on each host, but with all other demon processes still operating. The hosts were 3 VAX/11-780s connected with a 10Mbit Pronet. All measurements are in seconds.

Table 1 below contrasts remote access and local access. Note that for local files, the overhead introduced by IBIS for *open* is negligible. For local *reads*, the overhead rises for some reason with the size of the file. Placing IBIS into the kernel might eliminate that anomaly. For remote files, we need to distinguish whether IBIS is accessing the first file on the



(1) Reserve a port number A
(2) Execute connection starter with argument A
(3) Pass A and client's id to server creator via secure channel
(4) Fork a server process
(5) Reserve a port number B if authentication checking succeeds
(6) Pass B via secure channel back to client
(7) Establish new connection using A and B and verify

Figure 2: Two-way handshake via secure channel.

remote host, or subsequent ones. The initial access is quite expensive, since it involves setting up a dedicated server and a connection. Subsequent remote *open* operations are only about 4 times more expensive than a local *open*. The time for the initial remote *open* could be reduced by maintaining a dedicated connection per user, which is reused by every process owned by that user. A remote *read* is between 3 and 10 times more expensive than a local *read*. Again, placing the access transparency layer into the kernel should speed it up.

System call	File location	Open		Read		
		initial	non-initial	1K bytes	5K bytes	10K bytes
Standard calls	local	0.012	0.012	0.025	0.053	0.083
IBIS calls	local	0.013	0.013	0.026	0.094	0.136
IBIS calls	remote	1.787	0.053	0.078	0.377	0.884

Table 1: Performance of IBIS calls vs. standard system calls

Table 2 compares the performance of the UNIX 4.2 remote copy command (*rcp*) with the *cp* command linked with the IBIS library. The left half of the table shows copying times between a remote and the local host, the right half between 2 remote hosts. In the first case, *rcp* is about 50% slower than IBIS; in the second case about 70%. Note the file sizes chosen. According to [4] and [3], the average UNIX text file has about 250 lines, and with each line

having less than 40 characters, the average file consumes not more than 10K bytes.

Command	local <--> remote		remote <--> remote	
	10K bytes	20K bytes	10K bytes	20K bytes
IBIS cp	2.81	3.30	5.37	6.153
rcp	4.33	4.72	9.22	11.62

Table 2: Performance of IBIS cp versus rcp.

In summary, remote file access in IBIS is tolerably slower than local access. It runs somewhat faster than an an existing program (*rcp*) that performs remote access. Much room for improvement remains, since IBIS was implemented with the UNIX 4.2 IPC mechanism without efficiency considerations.

3. Location Transparency, Migration, and Replication

A number of UNIX networks provide access transparency, but no location transparency, migration, or replication. Examples are COCANET [5] and UNIX United [6]. LOCUS [7] provides location transparency and replication, but no migration. LOCUS also implements a centralized file access protocol which requires that each file access must first communicate with the "synchronization site" for that file to locate a valid copy. Thus, even if a node has a copy of a file, it must synchronize at a potentially remote site for opening and closing the file. In particular, the synchronization site of a replicated file is always remote. IBIS avoids both the bottleneck of a synchronization site as well as remote synchronization for local replicates. IBIS provides a more decentralized control for file access than LOCUS. The advantage of decentralization is more potential parallelism and better fault tolerance.

In the following, we describe IBIS' scheme for keeping a distributed and replicated file system consistent. We shall first discuss how files are treated, how replication and migration are controlled, and then present the directory level and the file lookup mechanism.

3.1. IBIS Files

An IBIS file has a unique file identifier or *fid*, which is a triple <host#, device#, inode#>. The fid of a file specifies on which host and device the file is located. The inode number uniquely identifies a file on a given device. Note that fids can be created on each host independently; no interrogation of a potentially remote synchronization site as in LOCUS is necessary.

An IBIS file is in one of 4 states: U, P, F, or S; compare Figure 3 for the complete state transition diagram. State U means that the file is a unique copy; there are no replicates anywhere. Thus, local operations on such a file incur no overhead. State P means that the file is the primary copy, and may have replicates in the network. All updates are performed on the

primary copy. Replicates are cached at other sites to speed up read accesses. If the primary copy is updated, the update broadcasts a signal that invalidates the cached copies. Furthermore, if a primary copy is updated, its state reverts to U, since there are no replicates. Thus, an update has the following properties: First, updates synchronizes at the file's home site; no other sites must be interrogated. Second, the cost of creating replicates is distributed to the machines with the replicates. Third, updates have the *contraction property*: They eliminate replicates and free space. Of course, the replicates will reappear if the updated file is accessed again remotely, but only at sites where the file is in use.

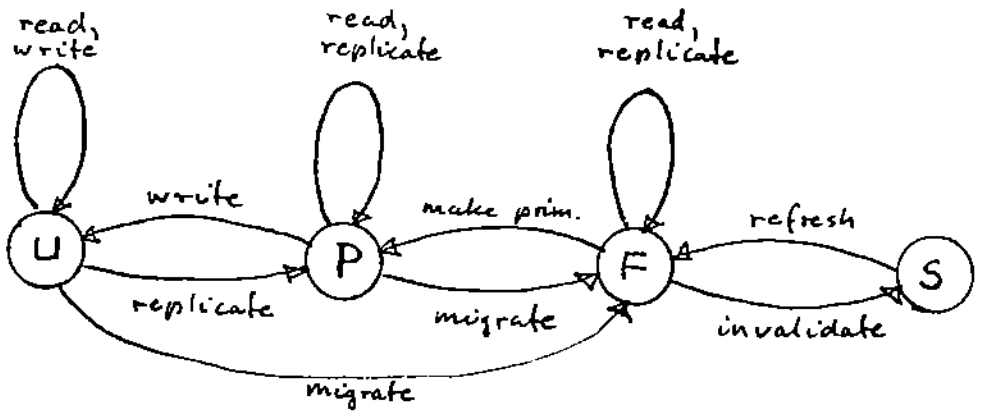


Figure 3: State transition diagram for IBIS files.

Replicated files are either in state F (fresh replicate) or S (stale replicate). State F means that no invalidation signal has been received from the primary copy, and the replicate is therefore assumed to be up to date. State S means the file is a replicate of an earlier version of the primary copy. Because of delays in propagating the invalidation signal, it is quite possible that a replicate is in state F, when it should be in state S. However, in the absence of timing channels among hosts, operations on a file are still serializable. If the delays of receiving the invalidation signal are intolerable for some applications, the best approach is to simply disallow replication of the affected files. The cost of instantaneous update of all replicates of a frequently changing file is much higher than remote access of a single copy.

Migration in IBIS means to change the site of the primary copy. Thus, migration is used mainly for speeding up write access, while replication improves read access times. Migration designates a fresh copy as the new primary. A second form is to migrate a file in state U by first copying it to a remote site and then marking the new file as the primary and the original as fresh. Note that after the next update, the old copy will disappear. Migration is an expensive operation, because it involves changing all directories containing the fid of the old primary. (Directories are discussed below.)

3.2. Replication and Migration Control

File replication pays off for those files that are read more often than written. Files and directories near the root of a hierarchical file system exhibit that property. In particular, directories experience much higher levels of reads than writes, and a high degree of replication undoubtedly improves system performance. IBIS therefore provides "*demand replication*" by default. Demand replication means that a replicate is cached locally whenever the corresponding primary or unique copy is read remotely. Thus, unless demand replication is disabled for a file, simply reading it generates a local copy of it. This strategy is quite appropriate near the root of the directory tree. At lower levels of the tree, sharing of subdirectories diminishes, while update traffic increases. Hence, less replication is desirable to improve performance. Note that because of the contraction property of updates, unused replicates will disappear after the next update. We therefore expect lower levels of the tree to automatically have little or no replication.

For reliability purposes, IBIS also provides "*forced replication*". Forced replication causes a certain number of replicates to be generated, even if no remote access occurs. A copy marked for forced replication refreshes itself as soon as the invalidation signal arrives. (Thus, replicate copies are "pulled" by the remote site, rather than "pushed" by the site holding the primary copy.)

It is also possible to totally disable replication for frequently updated files. This restriction is recorded in the unique copy, causing the state U to "stick" to it.

Automatic migration is more difficult to implement. IBIS will initially provide explicit commands for migration. As we gain more experience with network operating systems where processes seek out the hosts with the lowest loads, we plan to develop automatic placement and migration mechanisms.

3.3. Directories and Pathname Searching

Directories are implemented as files. They have the same state attributes and follow the same access protocol. A directory simply pairs character strings with fids of files (which may again be directories). Because the fid may belong to a remote file, and because there may be a local replicate, the directory must contain additional information, namely the fid of the local copy (if any). Thus, a directory implements the following 2 mappings:

character string --> primary fid
primary fid --> replicate fid

If the primary copy is on the local host, the primary fid and the replicate fid are identical. The replicate fid is undefined if there exists no local copy; such a directory entry is called a "*dead end*". Whenever directory lookup reaches a dead end, remote access is needed for locating the object. Fig. 4 below illustrates two hosts with two levels of a common directory tree which is partially replicated. Note that both mappings are in the same directory. An

alternative design decision would be to put all mappings from primary fid to replicate fid into a single table at each host.

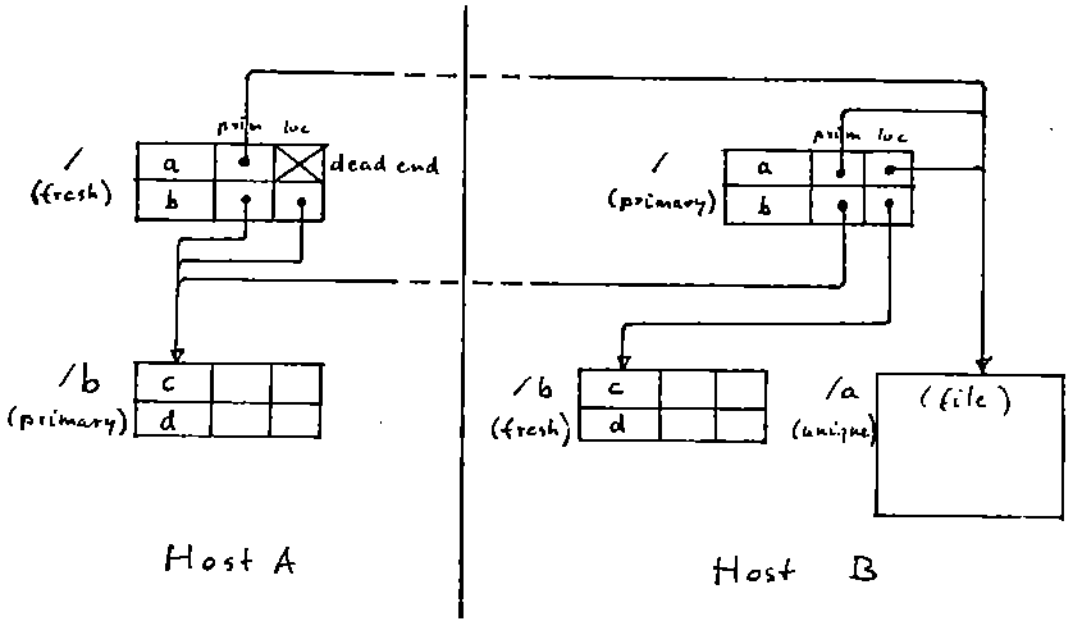


Figure 4: A replicated directory tree.

Replicated copies of directories are in general NOT identical, even if they are consistent. For instance, two directories on two machines listing a replicated file contain different replicate fids. When a stale directory replicate is refreshed, a simple copy operation is not sufficient. What must be updated is the mapping from character string to primary fid, but the mapping of primary fid to replicate fid must remain unchanged. Otherwise, the replicate of a whole subtree may be lost whenever the root of that subtree changes. For example, in Figure 4 consider what must be done in host A if the entry b in the root directory of host B is renamed to x. An exception is deletion: If a primary fid is deleted, then its mapping to the replicate fid should of course also be deleted.

It should now be clear how pathname searches proceed. Given a character string, a pathname search locates the primary fid and the replicate fid (if defined) of the file named by the character string. The replicate fid is desirable for local read access; the primary fid is needed for (potentially remote) write access. The search starts either with the current directory, or the file system root. To commence, one of these two directories is opened. This may be a local open of a unique or primary copy, or a remote open, or an open of a local replicate (possibly after restoring its state to F). The opened directory is searched for the first pathname component. If a match is found, the associated primary and replicate fids are retrieved. If the replicate fid exists, the search continues in the replicate, otherwise in the remote primary copy. The host number embedded in the primary fid indicates where the primary is

located. If the host with the primary is not reachable because of network partitioning, a broadcast may locate a replicate.

Much design work remains to be done. We need to develop robust algorithms for partitioning the file system in case of network failures, and for reconnecting it back together. If the network is partitioned, primary copies of files may no longer be reachable. In that case, a special protocol must designate one of the (hopefully available) replicates as a temporary primary, such that updates may proceed. For reconnecting the file system, we have developed a general three-way file merging technique that can merge two versions of a file with respect to a common ancestor [8]. A three-way file merge is reliable, provided there are no overlapping changes and the common ancestor of the two diverging primaries has been saved. The natural time to save the common ancestor is when a replicate is promoted to temporary primary status. Our merging technique merely requires that the user provide routines for extracting and comparing individual records for each type of file to be merged. Merging of directories and text files is done automatically. The merging technique generalizes to a $3(n-1)$-way merge, if the net is partitioned into $n \geq 2$ subnets.

4. Conclusions

The access transparency layer of IBIS is complete and operational. Using it is immediately addictive. After some initial experimentation, it becomes natural to access remote files, especially since there is not a single new command to learn. Directly editing a remote file is enormously more convenient than using remote login. Interactive programs with remote access (like screen editors) should be faster than remote login, because the high-bandwidth user interaction is done locally. Moreover, remote access commands can deal with files on several machines simultaneously, whereas remote login or a remote shell limits the user to one machine at a time. Finally, any existing program can be upgraded to remote access almost instantly.

Remote symbolic links have also proven to be quite useful. They permit users to simulate a directory tree spanning several machines. Remote symbolic links in conjunction with access transparency can almost provide location transparency. The only drawback is that the current directory must be on the local machine; it is impossible to change the current directory to a remote one. We plan to lift this restriction.

The access protocol for the location transparent level of IBIS has been carefully designed to avoid overhead for purely local operations. For example, local file creation, local read/write of a unique or primary copy, and local read of a replicate incur almost no overhead. In addition, the cost of making replicates is distributed, and the contraction property of updating assures that frequently written files experience a low level of replication. Demand replication, on the other hand, automatically replicates files and directories that are read-shared at many different hosts. File migration, finally, improves write access by automatically or semi-automatically moving files to the sites where they are written.

Appendix

The following manual pages describe the remote file access primitives of IBIS, and *rcsh*, a version of *csh* with remote I/O redirection and remote file name substitution.

References

1. Paris, Jehan Francois and Tichy, Walter F., "STORK: An Experimental Migrating File System For Computer Networks," pp. 168-175 in *Proceedings IEEE INFOCOM*, IEEE Computer Society Press (April 1983).
2. Comer, Douglas E., "Transparent Integrated Local and Distributed Environment (TILDE) Project Overview," CSD-TR-466, Technical Report, Purdue University, Department of Computer Science (1984).
3. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS (September 1982).
4. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software -- Practice and Experience* 9(1) p. 1-15 (Jan. 1979).
5. Rowe, Lawrence A. and Birman, Kenneth P., "A Local Network Based on the UNIX Operating system," *IEEE Transactions on Software Engineering* SE-8(2) (March 1982).
6. Brownbridge, D. R., Marshall, L. F., and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!," *Software -- Practice and Experience* 12 p. 1147-1162 (1982).
7. Popek, Gerald, Walker, Bruce, English, Robert, Kline, Charles, and Thiel, Greg, "The LOCUS Distributed Operating System," *Operating Systems Review* 17(5) p. 49-70 ACM, (Oct. 1983).
8. Tichy, Walter F., "The String-to-String Correction Problem with Block Moves," *ACM Transactions on Computer Systems* 2(4) (Nov. 1984). (to appear)

NAME

System calls with remote access:

access, chdir, chmod, close, creat, dup, dup2, fchmod, flock, fork, fstat, fsync, ftruncate, link, lseek, lstat, mkdir, open, read, readlink, rename, rmdir, stat, symlink, truncate, umask, unlink, write

Stdio functions with remote access:

clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fileno, fprintf, fputc, fputs, fread, fscanf, fseek, ftell, fwrite, getc, getchar, gets, getw, printf, putc, putchar, puts, putw, rewind, scanf, setbuf, setbuffer, setlinebuf, sprintf, sscanf, ungetc

Other library functions with remote access:

closedir, opendir, perror, popen, readdir, rewinddir, scandir, seekdir, telldir

SYNOPSIS

Same as those in UNIX Programmer's Manual (2) and (3).

DESCRIPTION

The functions listed in the first paragraph above mimic the system calls for file manipulation. The semantics are the same as described in the UNIX Programmer's Manual (2), except for accepting remote file names or remote file descriptors.

Open/creat a remote file or *dup* a remote file descriptor returns a remote file descriptor, which can be used later on in *read, write, lseek, fstat, dup,* ... in the same way as an ordinary file descriptor. Remote file descriptors are inherited upon *fork* (in libra.a), *vfork* and *exceve* as long as the program to be *exceve'd* also uses the remote access versions of the system calls/library functions.

When an error occurs remotely during a system call, the *errno* is copied to the external variable *errno* of the client process to indicate the error condition.

The sources of the functions listed in the second and third paragraphs are the same as those in libc.a. They have been linked to invoke the functions in the first paragraph instead of the standard system calls.

RESTRICTIONS

Remote access functions can cross machine boundaries only once. For instance, the following cannot be passed to remote access primitives: A remote directory entry which is a remote symbolic link to another machine; a remote symbolic link to a directory entry which in turn is a remote symbolic link.

System calls and library functions that have nothing to do with remote access are not included in *libra.a*. There are a few privileged system calls whose remote versions are not yet implemented. Such system calls include *chown, chroot, fchown, fcntl, mknod, mount* and *umount*.

FILES

/usr/ruan/ra/lib/libra.a

SEE ALSO

rcsh(1R), commands(1R)

intro(2), access(2), chdir(2), chmod(2), close(2), creat(2), dup(2), flock(2), fork(2), fsync(2), link(2), lseek(2), mkdir(2), open(2), read(2), readlink(2), rename(2), rmdir(2), stat(2), symlink(2), truncate(2), umask(2), unlink(2), write(2)

intro(3), directory(3), perror(3), popen(3), scandir(3)

intro(3S) and the whole section (3S)

NAME

rcsh – remote access version

SYNOPSIS

Same as *csh* but with remote access.

DESCRIPTION

Same as described in UNIX Programmer's Manual (1), except that remote file name [*host:*]*path* can be used in name substitution and I/O redirection.

Name substitution (globbing): the characters '*', '?', '[' and '{' in *path* are expanded on the machine *host* if *host:* is present or on the local machine if *host:* is omitted. However, '~' can be expanded only if *host:* is missing (i.e. as the first character of the whole file name). No substitution is allowed in the *host* part.

I/O redirection: standard input, standard output and diagnostic output can be redirected to remote files in the same way as to local files.

The other functions such as command interpretation, history substitution, alias substitution and variable substitution are the same as described in UNIX Programmer's Manual (1).

RESTRICTIONS

Rcsh supports remote file access but not remote command execution. The commands are always executed locally. The default input/output is the local terminal. For example, the following commands are legal:

cat mordred:... arthur:... | pg

ls -l mordred:... | grep ... > arthur:...

But

ls ... | mordred:grep ...

does not work.

There is no remote current working directory. Therefore,

cd *host:path* , or

pushd *host:path*

does not work.

BUGS

The built-in commands do not accept remote files as arguments or input/output redirection. For example,

echo "..." > mordred:... , or

exec ls arthur:...

does not work.

SEE ALSO

commands(1R), libra.a(3R)

csh(1), rsh(1)